The Edison Reference

Vivek Srikumar

January 30, 2013

Contents

1 Introduction **2**
1.1 What is *Edison*? . . . 2
1.2 License . . . 2

2 Getting Started **2**
2.1 Usage with Maven . . . 2
2.2 Installation without Maven . . . 2

3 Key concepts **2**
3.1 TextAnnotation . . . 3
3.2 Views . . . 3
3.3 Different types of views . . . 3

4 Feature Extraction **3**
4.1 Concepts . . . 3
4.2 Feature manifests and `.fex` definitions . . . 3
4.3 List of pre-defined features . . . 3
4.3.1 Bias feature . . . 3
4.3.2 Word features . . . 3
4.4 List of known transformers . . . 4
4.5 List of feature operations . . . 4

5 NLP Helpers **4**

6 Examples **4**
6.1 Basic examples . . . 4
6.1.1 Creating a `TextAnnotation` . . . 4
6.1.2 Accessing the text and tokens . . . 5
6.1.3 Accessing sentences . . . 6
6.2 Connecting to the Curator . . . 6
6.2.1 Adding views from the Curator . . . 7
6.3 Creating custom views . . . 8
6.4 Integrating your own feature extractors . . . 8

1 Introduction

1.1 What is *Edison*?

Edison is a Java library that is designed to help programming NLP applications by providing a uniform representation of various NLP annotations of text (like parse trees, parts of speech, semantic roles, coreference, etc.) This uniform representation allows for easy feature extraction that accesses several views. In addition to helping with feature extraction, the library also allows an easy way to access the Curator to further speed up development.

This manual is designed to help get started with *Edison* and its various features. *Edison* has been successfully used to facilitate the representation and feature extraction for several higher level NLP applications like semantic role labeling, coreference resolution, textual entailment, paraphrasing and relation extraction which use information across several views over text to make a decision.

1.2 License

2 Getting Started

2.1 Usage with Maven

To use *Edison* with Maven, you need to include the following in your `pom.xml`:

```
<repositories>
    <repository>
        <id>CogcompSoftware</id>
        <name>CogcompSoftware</name>
        <url>http://cogcomp.cs.illinois.edu/m2repo/</url>
    </repository>
</repositories>
```

Now, in the dependencies section of the pom file, add the following dependency:

```
<dependencies>
.
.
    <dependency>
        <groupId>edu.illinois.cs.cogcomp</groupId>
        <artifactId>edison</artifactId>
        <version>0.3</version>
        <type>jar</type>
        <scope>compile</scope>
    </dependency>
.
.
</dependencies>
```

2.2 Installation without Maven

3 Key concepts

In *Edison*, different annotations over text are called `Views`, each of which is a directed, labeled graph of `Constituents` (i.e., nodes) and `Relations` (i.e. directed, labeled edges). All the `Views` of a given text are managed by an object called a `TextAnnotation`.

One key assumption in *Edison* is that the views can be defined in terms of the tokens of the text. In other words, the `TextAnnotation` object fixes a tokenization for the text when an object is created and all the views are defined in terms of this tokenization.

3.1 TextAnnotation

3.2 Views

Edison stores all information about a specific annotation over text in an object called `View`. In its most general sense, a `View` is a graph whose nodes are labeled spans of text. The edges between the nodes represent the relationships between them. A `TextAnnotation` can be thought of as a container of views, indexed by their names.

The tokens are not stored in a `View`. The `TextAnnotation` knows the tokens of the text and each `Constituent` of every view is defined in terms of the tokens. A constituent can represent zero tokens or spans.

Sentences are stored as a view. In the terminology above, the `Constituents` will correspond to the sentences. There are no `Relations` between them. (The ordering between the sentences is not explicitly represented because this can be inferred from the `Constituents` which refer to the tokens.) So the graph that this View represents is a degenerate graph, with only nodes and no edges.

3.3 Different types of views

4 Feature Extraction

4.1 Concepts

- Feature extractors
- Feature input transformers
- Operations

4.2 Feature manifests and `.fex` definitions

4.3 List of pre-defined features

This section lists the set of pre-defined feature extractors along with their description and the `FeatureExtractor` that implements them.

4.3.1 Bias feature

The keyword *bias* in a `.fex` specification includes a feature that will always be present. This is useful to add a bias feature for binary classification.

4.3.2 Word features

The following list of feature extractors are operate on the last word of the input constituent. They are all defined as static members of the class `WordFeatureExtractorFactory`.

fex name	Feature Extractor	Description
capitalization	`capitalization`	Adds the following two features: One with the word in its actual case, and the second, an indicator for whether the word is captitalized
conflated-pos	`conflatedPOS`	The coarse POS tag (one of Noun, Verb, Adjective, Adverb, Punctuation, Pronoun and Other)
de-adj-nouns	`deAdjectivalAbstractNounsSuffixes`	An indicator for whether the word ends with a de-adjectival suffix. The list of such suffixes is in `WordLists.DE_ADJ_SUFFIXES`.
de-nom-nouns	`deNominalNounProducingSuffixes`	An indicator for whether the word ends with a de-nominal noun producing suffix. The list of such suffixes is in `WordLists.DENOM_SUFFIXES`.
de-verbal-suffixes	`deVerbalSuffix`	An indicator for whether the word ends with a de-verbal producing suffix. The list of such suffixes is in `WordLists.DE_VERB_SUFFIXES`.
gerunds	`gerundMarker`	An indicator for whether the word ends with an *-ing*.
known-prefixes	`knownPrefixes`	An indicator for whether the word starts with one of the following: *poly*, *ultra*, *post*, *multi*, *pre*, *fore*, *ante*, *pro*, *meta* or *out*
lemma	`lemma`	The lemma of the word, taken from the LEMMA view (that is, `ViewNames.LEMMA`)
nom	`nominalizationMarker`	An indicator for whether the word is a nominalization
numbers	`numberNormalizer`	An indicator for whether the word is a number
pos	`pos`	The part of speech tag of the word (taken from `ViewNames.POS`)
prefix-suffix	`prefixSuffixes`	The first and last two, three characters in the lower cased word
word	`word`	The word, lower cased
wordCase	`wordCase`	The word, without changing the case
date	`dateMarker`	An indicator for whether the token is a valid date

4.4 List of known transformers

4.5 List of feature operations

5 NLP Helpers

6 Examples

6.1 Basic examples

This set of examples goes over the basics of the Edison data structures. Recollect that different annotations over text are called `Views`, each of which is a graph of `Constituents` and `Relations`. The object that manages views corresponding to a single piece of text is called a `TextAnnotation`.

6.1.1 Creating a `TextAnnotation`

1. **Using the LBJ sentence splitter and tokenizer**

   The simplest way to define a `TextAnnotation` is to just give the text to the constructor. Note that in the following example, `text1` consists of three sentences. The corresponding `ta1` will use the sentence slitter defined in the Learning based Java (LBJ) library to split the text into sentences and further apply the LBJ tokenizer to tokenize the sentence.

```
1 String text1 = "Good afternoon, gentlemen. I am a HAL-9000 "
```

```
    + "computer. I was born in Urbana, Il. in 1992";

String corpus = "2001_ODYSSEY";
String textId = "001";

// Create a TextAnnotation using the LBJ sentence
// splitter and tokenizers.
TextAnnotation ta1 = new TextAnnotation(corpus, textId, text1);
```

2. **Using pre-tokenized text**

   Quite often, our data source could specify the tokenization for text. We can use this to create a `TextAnnotation` by specifying the sentences and tokens manually. In this case, the input to the constructor consists of the corpus, text identifier and a `List` of strings. Each element in the list will be treated as a sentence. This constructor assumes that the sentences in the list are white-space tokenized.

```
String textId2 = "002";

List<String> tokenizedSentences = Arrays.asList(
                                  "Good afternoon , gentlemen .",
                                  "I am a HAL-9000 computer .",
                                  "I was born in Urbana , Il. in 1992 .");
TextAnnotation ta2 = new TextAnnotation(corpus, textId2, tokenizedSentences);
```

3. **Other ways**

   The `TextAnnotation` class has several constructors, but the above examples cover the most important cases. Another important use case is to to create a text annotation using the Curator. This is covered in the section covering curator examples.

6.1.2 Accessing the text and tokens

Edison keeps track of the raw text along with the tokens it contains. So, we can get the original text using the function `getText()` and also the tokenized text using the function `getTokenizedText()`. The function `getToken(int tokenId)` gives us the tokens in the text.

```
// Print the text. This prints the raw text that was used to
// create the TextAnnotation object. In the case where the
// second constructor is used, the text is printed whitespace
// tokenized.
System.out.println(ta1.getText());
System.out.println(ta2.getText());

// Print the tokenized text. The tokenization scheme is
// specified by the constructor, which in the first example
// defaults to the LBJ tokenizer, and in the second one is
// specified manually.
System.out.println(ta1.getTokenizedText());
System.out.println(ta2.getTokenizedText());

// Print the tokens
for (int i = 0; i < ta.size(); i++) {
    System.out.print(i + ":" + ta.getToken(i) + "\t");
}
System.out.println();
```

6.1.3 Accessing sentences

Each `TextAnnotation` knows the views it contains. To get these, use the function `getAvailableViews()`, which returns a set of strings representing the names of the views it contains.

The following code prints all the available views in the `TextAnnotation` ta1 defined above. It then goes over each sentence and prints them.

```
System.out.println(ta1.getAvailableViews());

// Print the sentences. The Sentence class has many of the same
// methods as a TextAnnotation.
List<Sentence> sentences = ta1.sentences();

System.out.println(sentences.size() + " sentences found.");

for (int i = 0; i < sentences.size(); i++) {
    Sentence sentence = sentences.get(i);
    System.out.println(sentence);
}
```

6.2 Connecting to the Curator

The Curator acts as a central server that can annotate text using several annotators. With Edison, we can connect to the Curator to get those annotations and build our own NLP-driven application. *Edison* can be thought of as a Java client of the Curator.

The primary class we will use is the `CuratorClient`. To create a `CuratorClient`, we need to specify the host and the port of the curator server. There are two ways to access the Curator: (1) With the raw text and asking the curator to tokenize it for us, or, (2) Pre-defining the tokenization and asking the curator to respect it.

1. **Raw text**

   We could ask the Curator to provide us the tokenization and sentences. We would use this when we want to process raw text. The following example demonstrates this use case.

```
// This is the text we want to annotate.
String text = "Good afternoon, gentlemen. I am a HAL-9000 "
    + "computer. I was born in Urbana, Il. in 1992";

String corpus = "2001_ODYSSEY";
String textId = "001";

// We need to specify a host and a port where the curator server is
// running. Note: The following server does not exist and is used as
// an example.
String curatorHost = "curator.cs.uiuc.edu";
int curatorPort = 9090;

CuratorClient client  = new CuratorClient(curatorHost, curatorPort);

// Should the Curator be forced to update its cache if the exact text
// is already present? Unless you want to force the Curator to clean
// up its cached version of this text, set this to false.
boolean forceUpdate = false;

TextAnnotation ta = client.getTextAnnotation(corpus, textId, text,
```

```
22                                                    forceUpdate);
```

2. **Tokenized text**

   The other setting is when we have pre-tokenized text that we want to process with the different annotators that the Curator provides. In this case, we should ask the Curator to respect the tokenization that the `TextAnnotation` specifies. To do so, we need to use a different constructor for the `CuratorClient`.

```
1  // This is the text we want to annotate.
2  String textId2 = "002";
3
4  List<String> tokenizedSentences = Arrays.asList(
5                                    "Good afternoon , gentlemen .",
6                                    "I am a HAL-9000 computer .",
7                                    "I was born in Urbana , Il. in 1992 .");
8
9  TextAnnotation ta = new TextAnnotation(corpus, textId2, tokenizedSentences);
10
11 // We need to specify a host and a port where the curator server is
12 // running. Note: The following server does not exist and is used as
13 // an example.
14 String curatorHost = "curator.cs.uiuc.edu";
15 int curatorPort = 9090;
16
17 CuratorClient client  = new CuratorClient(curatorHost, curatorPort, true);
```

   Note: If this constructor is used to access the curator, then calling the function `getTextAnnotation` will trigger an exception.

6.2.1 Adding views from the Curator

Other than the creation of `TextAnnotation` objects, curator clients created with the two constructors described above are identical with respect to adding different views.

```
1  // Print the tokenized text
2  System.out.println(ta.getTokenizedText());
3
4  // Let's add the part of speech view and print it
5  client.addPOSView(ta, forceUpdate);
6
7  // The view is stored as 'ViewNames.POS'. The class ViewNames defines a
8  // set of standard names for different views.
9  System.out.println(ta.getView(ViewNames.POS));
10
11 // Add the named entity view and print it.
12 client.addNamedEntityView(ta, forceUpdate);
13 System.out.println(ta.getView(ViewNames.NER));
14
15 // Add the stanford dependency view and print the dependency tree
16 client.addStanfordDependencyView(ta, forceUpdate);
17 System.out.println(ta.getView(ViewNames.DEPENDENCY_STANFORD));
```

At present the CuratorClient has accessors for the following annotators: the Charniak and Berkeley parsers, coreference, easy-first dependency parses, named entities, verb and nominal SRL, Stanford constituent and dependency parsers and the Wikifier.

6.3 Creating custom views

6.4 Integrating your own feature extractors